CS611 Lecture 18 **Introduction to REC** 8 October 2003

Scribe: Tom Finley Lecturer: Radu Rugina

1 Introduction to REC Syntax

So far we have limited ourselves to the simple imperative language IMP. We reviewed three different types of semantics for this language. However, one of IMPs shortcomings is that there are no procedures! We now extend IMP with procedures. This language is called REC in Wynskel. In today's lecture, we explore the differences in this language from IMP.

We change IMP by taking out **while**, sequencing, and the assignment operator. We replace these with recursion, which is a million times worse. We have the following syntactic sets in the language:

$$\begin{array}{llll} x \in \mathbf{Var} & & n \in \mathbb{Z} \\ e \in \mathbf{Expr} & & f \in \mathbf{Fvar} \end{array}$$

Notice that we no longer have an distinction between **Aexp** and **Bexp**, as we shall see here. We define expressions as follows.

$$e ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid \mathbf{ifz}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 \mid f_i(e_1, \ldots, e_{a_i})$$

Much is familiar in the above list from **Aexp**, but for the additions, note that $\mathbf{ifz}\, e\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$, the **ifz** means specifically "if zero," that is if e evaluates to zero then the expression is e_1, and if e evaluates to nonzero then the expression is e_2. The $f_i(e_1, \ldots, e_{a_i})$ expression denotes a call to function f_1, which accepts a_i input arguments. Note that there is no comparison expression $e_1 \leq e_2$ or $e_1 = e_2$ in this syntax. A test "`if` $(e_1 = e_2)$ `then`..." can be easily modeled as "`ifz` $(e_1 - e_2)$ `then`...". The less-than comparison can also be modeled in REC using functions (see below).

A procedure declaration d defines k functions:

$$\begin{array}{rcl} d & ::= & f_1(x_{11}, x_{12}, \ldots, x_{1a_1}) = e_{f_1} \\ & & \vdots \\ & & f_k(x_{k1}, x_{k2}, \ldots, x_{ka_k}) = e_{f_k} \end{array}$$

Function f_i accepts a_i arguments and returns the result of evaluation expression e_{f_i}. In other words, e_{f_i} is the body of function f_i. We allow procedures to be either directly or mutually recursive.

A program p consists of a declaration d plus an expression e to be evaluated, specifically:

$$p ::= d\ e$$

2 Examples

The following $S(x)$ is the *sign* function that returns 1 if $x > 0$, -1 if $x < 0$, and 0 if $x = 0$. The following program has two function definitions, and then evaluates $S(17)$, which should return 1. The $f(x, y)$ compares $x \geq y$ for nonnegative x, y in that $f(x, y) = cond(x \geq y, 1, -1)$.

$$\begin{array}{l} S(x) = \mathbf{ifz}\, x\, \mathbf{then}\, 0\, \mathbf{else}\, f(x, 0 - x) \\ f(x, y) = \mathbf{ifz}\, x\, \mathbf{then}\, 1\, \mathbf{else}\, (\mathbf{ifz}\, y\, \mathbf{then}\, -1\, \mathbf{else}\, f(x-1, y-1)) \\ S(17) \end{array}$$

We can use similar functions to model comparisons $e_1 \leq e_2$. Consider the function:

$$\begin{array}{l} CMP(x, y) = \mathbf{ifz}\, (y - x)\, \mathbf{then}\, 0\, \mathbf{else}\, f(y - x, x - y) \\ f(x, y) = \mathbf{ifz}\, x\, \mathbf{then}\, 0\, \mathbf{else}\, (\mathbf{ifz}\, y\, \mathbf{then}\, 1\, \mathbf{else}\, f(x-1, y-1)) \end{array}$$

Then, a test "`if` $(e_1 \leq e_2)$ `then`..." can be modeled as "`ifz` $CMP(e_1, e_2)$ `then`...". From now on, we will use "`if` $(e_1 \leq e_2)$" to denote "`ifz` $CMP(e_1, e_2)$", where CMP is the above function.

The following is an example of a program that uses a function to compute the factorial of 6.

$fact(n) = \textbf{ifz}\ n\ \textbf{then}\ 1\ \textbf{else}\ n * fact(n-1)$
$fact(6)$

Next is a function *gcd* that computes the greatest common divisor of two numbers x and y with the help of two support functions f_1 and f_2. Function f_1 returns the maximum of its arguments, and function f_2 returns the absolute value of the difference of its arguments.

$f_1(x, y) = \textbf{ifz}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y$
$f_2(x, y) = \textbf{ifz}\ x \leq y\ \textbf{then}\ y - x\ \textbf{else}\ x - y$
$gcd(x, y) = \textbf{ifz}\ (x - y)\ \textbf{then}\ x\ \textbf{else}\ gcd(f_2(x, y), f_1(x, y))$
$gcd(6, 9)$

3 Semantics

There are two possible semantics to REC: call by value, and call by name. Intuitively, the two semantics differ in when the expressions passed to functions are evaluated. In call by value, the argument expressions are first evaluated to a number and that number is passed into the function. In call by name, the expression itself is passed into the function and is evaluated if necessary.

In an imperative language, call by name would lead to problems with state changes. Since there is, essentially, no variable state in REC in that there is no assignment operator, this is not a problem. However, the semantics are not equivalent. Suppose there is an expression that falls into an infinite loop, passed as an argument to a function that ignores that argument. In call by value the expression will be evaluated immediately (and so hang) whereas in call by name it will be evaluated when needed (in this case never) so the function will not hang (provided all used argument expressions terminate).

3.1 Large Step Operational Call-by-Value Semantics

Here we define the large step evaluations of an expression to a number, $e \to n$.

$$\frac{}{n \to n}$$

$$\frac{e_1 \to n_1 \quad e_2 \to n_2}{e_1 \oplus e_2 \to n} \text{ where } n = n_1 \oplus n_2 \text{ for } \oplus \in \{+, -, *\}$$

$$\frac{e \to 0 \quad e_1 \to n_1}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to n_1}$$

$$\frac{e \to n \quad e_2 \to n_2}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \to n_2} \text{ where } n \neq 0$$

$$\frac{e_1 \to n_1 \quad \cdots \quad e_{a_i} \to n_{a_i} \quad e_{f_i}[x_{i1} \to n_1, \ldots, x_{ia_i} \to n_{a_i}] \to n}{f_i(e_1, \ldots, e_{a_i}) \to n}$$

Notice, we have no variable space, and no assignment. Variables are only "assigned" in the sense that expressions are evaluated to values, and these passed as values to functions that substitute for various values within the function expression e_{f_i}. Getting in a stuck configuration is treated as an error.

3.2 Denotational Call-by-Value Semantics

The denotational semantics take the form $[\![e]\!]\rho\varphi$, where $\rho : \mathbf{Var} \to \mathbb{Z}$ is our variable environment that maps variables to values, and φ is our function environment $\varphi : (\mathbb{Z}^{a_1} \to \mathbb{Z}_\perp) \times \cdots \times (\mathbb{Z}^{a_k} \to \mathbb{Z}_\perp)$. Whenever we get to $\perp$, we stop evaluation and propagate $\perp$.

$$
\begin{array}{rcl}
[\![n]\!]\rho\varphi & = & \lfloor n \rfloor \\
[\![x]\!]\rho\varphi & = & \lfloor \rho(x) \rfloor \\
[\![e_1 \oplus e_2]\!]\rho\varphi & = & [\![e_1]\!]\rho\varphi \oplus_\perp [\![e_2]\!]\rho\varphi \\
[\![\mathbf{ifz}\; e \;\mathbf{then}\; e_1 \;\mathbf{else}\; e_2]\!]\rho\varphi & = & \begin{cases} [\![e_1]\!]\rho\varphi & \textit{if } [\![e]\!]\rho\varphi = \lfloor 0 \rfloor \\ [\![e_2]\!]\rho\varphi & \textit{if } [\![e]\!]\rho\varphi = \lfloor n \rfloor \textit{ and } n \neq 0 \\ \perp & \textit{if } [\![e]\!]\rho\varphi = \perp \end{cases} \\
[\![f_i(e_1, \ldots, e_{a_i})]\!]\rho\varphi & = & \begin{cases} \varphi_i([\![e_1]\!]\rho\varphi, \ldots, [\![e_{a_i}]\!]\rho\varphi) & \textit{if } [\![e_1]\!]\rho\varphi \neq \perp, \ldots, [\![e_{a_i}]\!]\rho\varphi \neq \perp \\ \perp & \textit{otherwise} \end{cases}
\end{array}
$$

In the above rules, note that we treat $\varphi = (\varphi_1, \ldots, \varphi_k)$, and that $\oplus_\perp$ is a construction to let arithmetic operators allow $\perp$ as an operand. This facilitates the propagation of $\perp$ values (for non-termination). For example, in the case of addition:

$$
n_1 +_\perp n_2 = \begin{cases} \lfloor n'_1 + n'_2 \rfloor & \textit{if } n_1 = \lfloor n'_1 \rfloor,\, n_2 = \lfloor n'_2 \rfloor \\ \perp & \textit{otherwise} \end{cases}
$$